ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Título del proyecto:

ETERNITY 2

Alumno: Pablo José Pinilla Sánchez
Tutor: Miguel Pagola
Pamplona, a 6 de Septiembre de 2010

1 INTRODUCCION

Con la realización de este proyecto se pretende informatizar la resolución de un puzzle en concreto, el Eternity II, aplicando técnicas de inteligencia artificial. Estas técnicas se pueden luego generalizar a la gran mayoría de los puzzles introduciendo ligeros cambios. Para ello se hará una pequeña introducción de cada uno de los dos campos que se tocan, los puzzles y la inteligencia artificial, para ir entrelazándolos hasta llegar al punto en que hemos desarrollado el proyecto.

1.1 INTRODUCCION HISTORICA DE LOS PUZZLES

El primer puzzle, de John Spilsbury

El puzzle se inventó en 1767 en Inglaterra por un fabricante ingles de mapas, John Spilsbury. Este serró un mapa de Europa por países, y su función era meramente educacional, para las clases de Geografía. El efecto de la sierra sobre la madera al ser cortada hizo que se denominase al puzzle originalmente jig saw (sierra de vaivén en ingles). Los puzzles que se hacían en esa época eran puramente educativos, pudiendo ser desde imágenes de la Biblia a tablas de matemáticas.

El primer puzzle que se comercializó fue "The Smashed Up Locomotive.", en 1880 creado por Milton Bradley, que consistía en unas piezas que al montarlas mostraba una locomotora entera con sus partes, desarrollando la misma función educativa que sus predecesores.

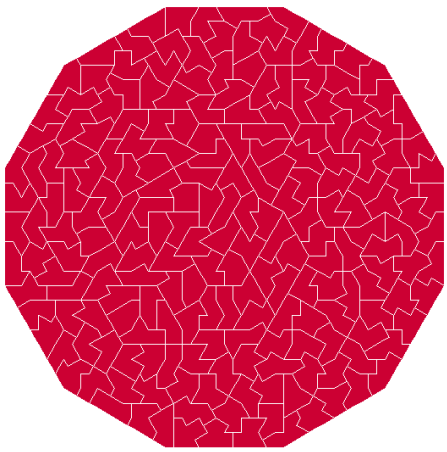
Posteriormente se introdujeron en los Estados Unidos a principios de siglo XX, tomando motivos más pictóricos y utilizando los puzzles por primera vez para el entretenimiento. Eso si, seguían siendo exclusivos para la gente de clase alta, debido a sus altos precios.

Por fin los precios bajaron tras la generalización del cartón como material principal en su fabricación, durante la gran depresión, lo que causó el gran esplendor de este juego y su inclusión en todas las clases sociales. Incluso ya lo utilizaron de reclamo de otros productos.

Desde entonces los puzzles han sido una parte importante del entretenimiento de todo el mundo, y ha evolucionado hacia todo tipo de formas, tamaños, motivos…

1.1.1 ETERNITY I Y ETERNITY II

Con esto llegamos a la serie de puzzle en la que esta basado este proyecto. El primer Eternity salió en 1999 y estaba formado por 209 piezas y un tablero con forma de dodecágono regular. El creador del puzzle ofreció un millón de libras para el que resolviese el puzzle. Fue uno de los puzzles más exitosos en el Reino Unido. Al final se encontraron dos soluciones diferentes al puzzle.



Este nuevo Eternity en cambio es un tablero cuadrado de 16x16, 256 piezas, todas cuadradas y dividas en triángulos de diferentes colores. El fin de este puzzle es hacer coincidir todos los colores, siendo todas las piezas únicas. Este puzzle salió en Julio de 2007 y se premia con 2 millones de dólares a la primera persona que lo resuelva. Cada año hace una convocatoria para ver si alguien lo ha conseguido, y a día de hoy aún no se ha repartido ese dinero.

1.2 INTRODUCCION HISTORICA DE LA INTELIGENCIA ARTIFICIAL

La inteligencia artificial como tal se empezó a desarrollar en 1943 con el trabajo de McCullogh y Pitts. La meta de la inteligencia artificial en esta época fue el desarrollo de métodos y algoritmos que permitan comportarse a un ordenador de manera inteligente, que puedan pensar de forma “humana”. En 1949 Hebb implementó la base de las redes neuronales. En 1950 Shannon desarrollo el primer programa que implementaba un ajedrez. Ese mismo año Turing publicó el artículo “Computing Machinery and Intelligence”, y propuso lo que se conoce como Prueba de Turing, para determinar si una maquina era inteligente o no, por lo que se le considera el padre de la Inteligencia Artificial. En 1956 se acuñó el término inteligencia artificial. En 1959 se crea el lenguaje LISP, con el que se hacen la mayoría de sistemas expertos en la actualidad.

Durante la década de los 60 y parte de los 70, los científicos se centraron en la representación del conocimiento, a base de representaciones de problemas, búsquedas y heurísticas centradas en puzzles o recuperar información. En 1962 se usaron las redes neuronales para el aprendizaje y el reconocimiento de patrones y un año después se codificaron los primeros programas que permitían la visión artificial.

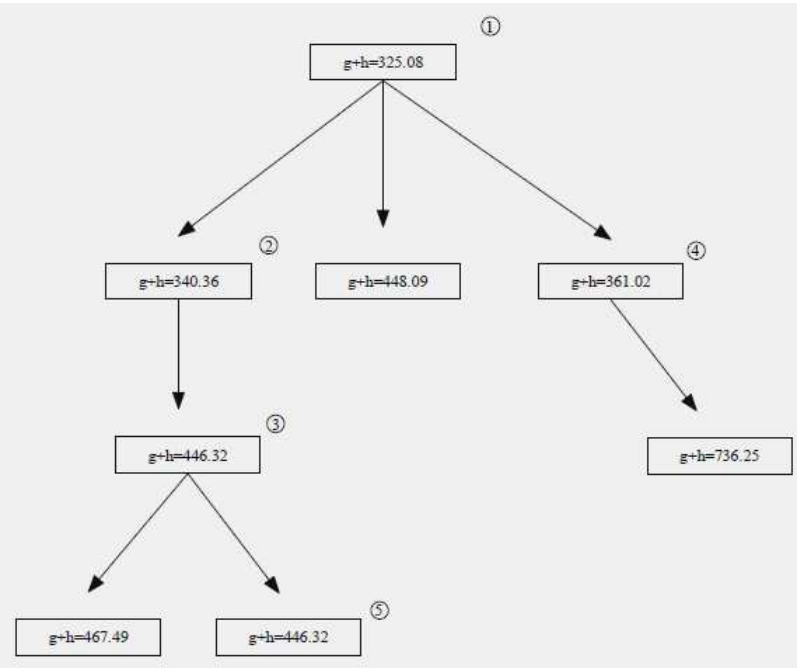
A partir de la segunda parte de los 70, los especialistas dejaron de lado los “problemas juguete” para enfrentarse a “problemas reales”. Para ello necesitaban introducir más conocimiento sobre el dominio de la aplicación. En 1980 se resuelven problemas de configuración de ordenadores y medicina por medio de sistemas expertos. En 1986 se desarrollan las redes multicapas y el algoritmo de retropropagación de error.

Desde ahí al momento actual se han desarrollado los lenguajes orientados a objetos, Garry Kasparov, campeón mundial de ajedrez, perdió contra el ordenador Deep Blue. Además de eso, son parte importante de programas aplicados de economía, medicina, ingeniería, sin contar con las aplicaciones de software, buscadores y de entretenimiento como videojuegos.

2 BUSQUEDA A*

Debido a la naturaleza del Eternity 2, la mejor forma de enfocar el problema era con las técnicas de los años sesenta y setenta, cuando se centró el estudio de la inteligencia artificial a los puzzles. En este caso, el puzzle se puede tratar como un árbol de nodos, siendo cada una de las situaciones que se dan, de las piezas que faltan por poner y la posición del resto de piezas un nodo propio. Para ello aplicaremos el algoritmo de búsqueda A*

Este algoritmo fue desarrollado por Hart en 1961. Esta basado en la búsqueda optimal. El algoritmo optimal consiste en ordenar la lista de nodos abiertos (los nodos abiertos son aquellos que aún pueden ser o conducir a la solución final) y coger siempre primero el que más se ajusta a nuestras necesidades. La búsqueda A* gana eficiencia al hacer estimaciones de la proximidad de los estados a la solución. Para conseguir esto asignamos a cada nodo un valor. Seguidamente la búsqueda siempre tratara de seleccionar el primero que más se ajusta, como en la optimal, pero siempre en función de esta heurística y el coste del camino por el que ha llegado. Por tanto la meta de este algoritmo es hallar la heurística perfecta, aquella que haga encontrar siempre el camino correcto de manera más directa.



Ejemplo de Árbol A* desarrollado

Esta heurística (la llamaremos f(x)), está formada por el coste del camino (lo llamaremos g(x)), que es el coste que hay desde el nodo inicial hasta el nodo actual; y una estimación heurística de la distancia a la meta (la denominaremos por h(x)).

El procedimiento es el siguiente, primero tenemos que declarar una serie de variables. Estas son abiertos, actual, cerrados y nuevos sucesores. Actual sirve para almacenar el nodo actual, y se inicia con una lista vacía. Cerrados es una lista que contiene los nodos ya analizados, iniciándola con la lista vacía. La variable abiertos es una lista de los nodos generados aún no analizados. Se inicia con el nodo inicial, que es el nodo que contiene el estado inicial. Y por último está nuevos sucesores, que es la lista de sucesores del nodo actual. También se inicializa con la lista vacía.

Tras esto nos ponemos a ejecutar el algoritmo:

MIENTRAS que **ABIERTOS** no esté vacía
Hacer **ACTUAL** el primer nodo de **ABIERTOS**
Hacer **ABIERTOS** el resto de **ABIERTOS**
Poner el nodo **ACTUAL** en **CERRADOS**
SI el nodo **ACTUAL** es un final, ENTONCES
Devolver **ACTUAL** y terminar
SINO
Hacer **NUEVOS-SUCESORES** la lista de sucesores de **ACTUAL** que no están en **ABIERTOS** ni en **CERRADOS** un nodo con el mismo estado y menor coste
Hacer **ABIERTOS** la lista obtenida de añadir **NUEVOS-SUCESORES** al final de **ABIERTOS** y ordenados sus nodos por sus heurísticas
Fin si
Fin mientras
SI **ABIERTOS** está vacía ENTONCES devolver NIL
Fin si

Empezando en el nodo inicial, siempre se mueve dentro de una lista con prioridad, abiertos. Cuanto menor sea el valor de heurística (f(x)) de un nodo, mayor será la prioridad de este dentro de la cola. En cada paso por el bucle cogemos el nodo con la menor heurística de la cola y lo quitamos, y lo introducimos en la lista de cerrados. Desde este nodo sacamos los sucesores, en los que tenemos que calcular los nuevos valores de f y h para cada uno de ellos. Estos sucesores se comprueban si ya están en la lista de abiertos y de cerrados, y los que no se introducen en abiertos y se vuelve a ordenar. Este proceso lo repetimos hasta dar con la solución o hasta que

nos quedemos sin nodos (si el problema tiene solución esto no se puede dar ya que el algoritmo es completo).

Las ventajas del A* es que utilizando una heurística admisible, podemos decir que es un método completo, porque si hay una solución la encuentra siempre, y es minimal, que esa solución la encuentra con la menor cantidad de nodos analizados. Si tenemos que r es el factor de ramificación, y p la profundidad de la solución, entonces la complejidad en espacio y tiempo será de $O(r^p)$.

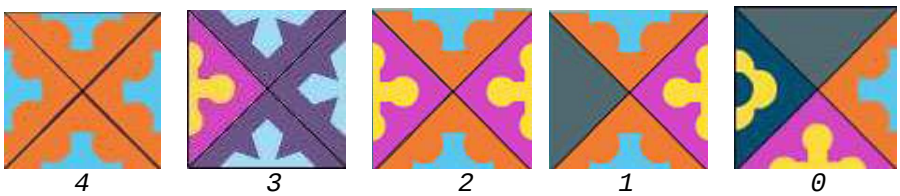
2.1 CASO CONCRETO

En este proyecto desarrollamos tres formas diferentes, para cada uno de los casos. Los dos primeros son sin heurística y el otro con una heurística que ahora vamos a mostrar.

El primer supuesto lo hacemos para el caso que no haya heurística y siempre sea el mismo camino. Esto lo conseguimos utilizando siempre el algoritmo de primero el mejor. Consiste en añadir los nuevos sucesores en el orden que han sido resueltos y coger siempre el primero y desarrollarlo.

El segundo supuesto lo hacemos también para el caso que no haya heurística, pero en esta ocasión cogeremos un nodo de nuevos sucesores al azar como el nodo a desarrollar.

El último supuesto es teniendo en cuenta una heurística propia. Hay que tener en cuenta que esta heurística no cuenta con el desarrollo de la heurística del camino, sólo con la heurística de la última ficha puesta. Se construye a partir del número de colores repetidos que tenga una ficha. Gráficamente la heurística es esta:

Heurística:  *4* *3* *2* *1* *0*

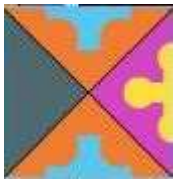
Consideramos como ficha fácil aquella con mayor heurística. Por lo que intentaremos siempre poner las difíciles primero, las que no tienen colores en común.

3 IMPLEMENTACION

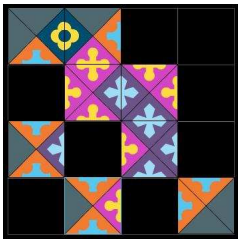
3.1 MODELO

Para la ejecución de cualquier proyecto, uno de los pasos más importantes es conseguir modelarlo de la forma más fiable a la realidad y más cercana al lenguaje de programación en el que lo vayas a representar. En nuestro caso, nos vamos a basar en tres estructuras de fácil entendimiento y desde las cuales implementaremos el resto de funciones.

Todo el proyecto esta implementado en el lenguaje C, debido a que era el que más flexibilidad y mayor rendimiento nos daba en la memoria y cálculo de nodos, ya que podemos optimizar el tamaño de la variable al máximo, y como en estas búsquedas abren tantos nodos, ocupan una gran cantidad de memoria.



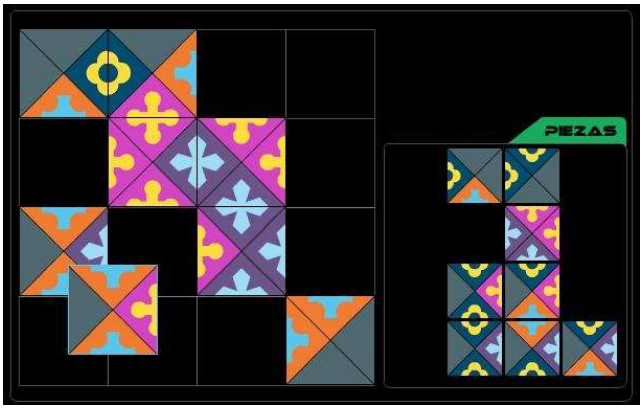
Empezamos con la unidad más elemental, las fichas del puzzle. En el Eternity II las fichas son cuadradas, dividas en cuatro zonas, cada una de ellas con un color propio. En nuestro modelo lo primero será identificar cada una de las fichas. A cada uno de los colores le asignaremos un número propio, que es algo que un ordenador puede procesar con mayor facilidad. Esto lo guardaremos en una tabla de cuatro valores, siempre guardándolos de la forma NESO, empezando por el vértice superior y en sentido de las agujas del reloj. Luego, para evitar tener que hacer la operación de giros, cuadriplicaremos la ficha y las guardaremos como ONSE y demás formas. Para distinguirlas tendremos un nuevo campo que sirva para decir en qué posición se encuentra la N. También hemos añadido un campo para agilizar los cálculos, uno en el que nos dice a cuantos bordes está pegada la ficha, para saber si es borde o esquina sin tener que hacer cálculos cada vez que vamos a poner una ficha, y así no poner fichas que son del borde dentro del tablero. Por último, en la última versión introducimos el campo de heurística de cada una de las fichas.



La segunda estructura irá implícita en la tercera y será el campo más importante de ella. Esta estructura es el tablero, que mostrará el tablero en un instante determinado. Las dimensiones del tablero serán de (filas + 2) x (columnas + 2), esto se da porque añadimos al exterior del tablero una nueva franja de fichas imaginarias formadas del color del borde. Para hacerlo más liviano y así tener que

ocupar menos memoria, lo que hacemos será guardar el puntero a la ficha, con ello conseguimos no tener que clonar la ficha cada vez que tengamos que introducirla, tener siempre una lista de todas las fichas y cuando tengamos que liberar sea mucho más fácil.

La tercera estructura, el nodo, es la más importante, la que engloba todos los campos y les da un sentido para el programa. Es la unidad con la que va a estar operando el A*. Está compuesta por el tablero, por una tabla de las piezas que no han sido utilizadas aún y un indicador que nos dice cual es la siguiente posición del tablero que se tiene que rellenar. Resumiendo, esta es la estructura que nos dice en qué momento de la evolución del puzzle estamos, desde la cual podemos desarrollar un nuevo árbol de decisiones o incluso analizarlo como nodo final.



La última estructura es la pila de nodos. Es necesaria para poder almacenar todos los nodos y que sea una variable local y no una variable global, que siempre hay que intentar dejarlas de lado. Estará compuesta con una lista de los nodos y el número de nodos que hay introducidos en ese momento. Tendrá las operaciones típicas de las pilas (sacar elemento, meter elemento y ver la cima).

3.2 GENERACION DE FICHAS

Tal y como hemos estructurado el programa, está implementado como los programas orientados a objetos. Esto hace que tengamos algunas de las funciones típicas de otros lenguajes, java, visual… Eso si, al ser C, tenemos que escribirlas nosotros, por eso tenemos la de NewFichatab, o la de BorradoFich. En la primera le pasas todos los datos para conseguir rellenar los campos de las fichas, y en la otra le pasas una lista de fichas y libera toda la memoria ocupada.

En este apartado tenemos las funciones que operan con las fichas, como sacarlas o introducirlas en ficheros, o producir las fichas equivalentes. Primero vamos a hablar de las que operan con ficheros. Estas son Sacafichas, MeteFichas y

TransFiche, una función especial para un fichero en concreto. SacaFichas lo que hace es leer un fichero donde están almacenadas las fichas en modo línea de texto, y las va creando en la memoria con la función de NewFichatab:

```
while(fgets(linea, 25, file)!=NULL)
{
	aux++;
	tabla=(fichatab **)realloc(tabla,aux*sizeof(fichatab *));

	//sscanf para convertir el carácter a número
	sscanf(linea,"%d %d %d %d %d %d",&i1,&i2,&i3,&i4,&i5,&i6);
	tabla[aux-1]=NewFichtab(i1,i2,i3,i4,i5,i6);
}
```

MeteFichas hace el proceso inverso, con las fichas que existen en memoria, las vuelca en un fichero al que le pasas el nombre. El formato en el que se encuentran en el fichero es: ”2 3 0 0 1 13”, siendo el primero el identificador de la ficha, el segundo la orientación, los cuatro siguientes los colores y en el caso de heurística un campo más. Las tres funciones anteriores tienen su equivalente con heurística, simplemente pasándole un campo más. La última función, TransFichero, sirve para cuando pasan el fichero original de las 256 fichas. Este fichero sólo tiene los colores, por lo que la función genera un identificador y la orientación (por defecto siempre norte).

Por último están las funciones para generar las fichas equivalentes por medio de giros. Como comentamos en la sección anterior, si tenemos las fichas ya generadas como fichas independientes, no tendremos que hacer cálculos para ver si se ajustan a la posición. Estas funciones son FichEquivalentes y TodFichEquivalentes. La primera le pasas una ficha y te devuelve una lista con la original y las generadas por cada uno de los giros. A la otra se le pasan las fichas en una tabla, y va una por una llamando a la función anterior, después las agrupa todas en una lista y es el resultado que devuelve. Estas dos funciones se utilizan la primera vez que le pasan un fichero, que no tiene todas las fichas, en posteriores simulaciones sólo se utiliza la de sacar fichas.

3.3 PREPARACION DE NODOS

En este bloque tenemos que preparar todas las funciones básicas necesarias a nivel de nodo para poder en un futuro operar con ellos en las búsquedas que hagamos. Podemos distinguir dos tipos de funciones, uno de control y otro de simulación.

El primer bloque, el de control, es el que sirve para visualizar o controlar el estado de un nodo o de una de sus características. Este bloque lo componen las tres funciones de visualizar y el que mira si un nodo es final. Empezamos por EsFinal. Esta función mira si hay fichas aún por utilizar y devuelve un número que actúa como booleano. En el caso que aún queden, devolverá 0, falso, y en el caso contrario 1. La siguiente a analizar es VisuInutil, en esta mostrará por pantalla la posición de memoria, el identificador y la orientación de cada ficha que este en la lista de fichas no utilizadas en el nodo dado. Por último están las que muestran el tablero de un nodo, VisuTable y VisuTable2D. El primero te muestra las fichas en este formato, mostrando toda la información de las fichas por orden de abcisas y ordenadas, pero dificultoso de leer y comprobar si de verdad funciona bien los ajustes entre fichas:

```
( 5, 5)==   6 4  3  1  1  2
( 5, 6)==   ? ?  ?  ?  ?  ?
```

El segundo en cambio te muestra sólo las posiciones, los colores de las fichas en dos dimensiones, de tal forma que te haces una idea de cómo se encuentra el tablero mucho más visualmente. Además hay que tener en cuenta que muestra las fichas imaginarias:

```
    0      3      2      3      1      3     ??      0
  0  0   0  2   2  3   3  1   1  2   2  1  ?? ??   0  0
    0      5      1      1      1      1     ??      0
```

En ambos casos, si no se han rellenado las posiciones aún con fichas, saldrán interrogantes como muestran los ejemplos puestos anteriormente.

Ahora vamos con el bloque de funciones de simulación. Como explicamos en la sección anterior, el proyecto necesita de las funciones típicas de los lenguajes orientados a objetos. Por lo que ahí tenemos las tres primeras, la de NodoInicial y ElimNodo, el constructor y el destructor, y a su mismo nivel ClonNodo. El primero marca el nodo inicial, reservando memoria, creando el tablero vacío (teniendo en cuenta las fichas imaginarias y el resto marcándolas punteros a NULL), todas las fichas dentro de la lista de fichas inutilizadas y el target marcando a la primera posición que debe ser rellenada.

```
nuevo->tab=(fichatab ***)malloc((N+2)*sizeof(fichatab **));
for(i=0;i<=N+1;i++){
	nuevo->tab[i]=(fichatab **) malloc ((M+2)*sizeof(fichatab *));
	for(j=0;j<=M+1;j++){
		if(i==0||i==N+1||j==0||j==M+1)
			nuevo->tab[i][j]=borde;
		else
			nuevo->tab[i][j]=NULL;
	}
}
```

Creación del tablero

ClonNodo hace la misma función, al pasarle un nodo lo duplica exactamente pudiéndolo modificar sin perder la información del primero. Y por último, está el destructor, ElimNodo, que al pasarle un nodo libera toda la memoria que ocupa él, su tablero y su tabla adicional. De la función de clonar nodos se derivan dos funciones necesarias para su funcionamiento, CloneTablero y CloneTabla. El primero clona un tablero de un nodo a otro, reservando memoria y asignando las posiciones. El segundo clona un doble puntero de fichas. Servirá para el momento que necesitemos duplicar las fichas restantes y cargarnos punteros a esas fichas, pero tener siempre una tabla original donde se encuentren todas esas direcciones y no la tengamos que estar creando cada vez que hagamos un nuevo nodo.

```
while(tabla[i]!=NULL)
{
                              //uno mas x empezar en 0
       aux=(fichatab **)realloc(aux,(i+1)*sizeof(fichatab*));
       aux[i]=tabla[i];
       i++;
}
aux=(fichatab **)realloc(aux,(i+1)*sizeof(fichatab*));
aux[i]=NULL;
```

Bucle de creación de una nueva tabla

La última función relacionada con tablas es BorraEquivalentes. En esta función le pasas la tabla de fichas y la ficha que ha sido elegida y se borran todas aquellas que tengan el mismo identificador, además de reajustar su memoria interna. Al estar ordenadas por el identificador, cogemos la primera que tenga ese identificador y borramos esa y las tres posteriores en la tabla.

```
while(tabla[i]->id!=elegida->id)
{
       i++;
}
while(tabla[i+4]!=NULL){
       tabla[i]=tabla[i+4];
       i++;
}
tabla[i]=NULL;
tabla[i+1]=NULL;
tabla[i+2]=NULL;
tabla[i+3]=NULL;
tabla=(fichatab**)realloc(tabla,(i+1)*sizeof(fichatab *));
```

Por último, se encuentran AvancetargetEsc e InsertaFicha. La primera al pasarle un nodo avanza el target a la siguiente posición. En este caso rellenaremos las fichas de forma escalonada, de izquierda a derecha y de arriba abajo. La otra función sirve

para insertar la ficha en la posición, borrar las fichas equivalentes y pasar a la siguiente posición.

3.4 PREPARACION DE BUSQUEDAS

Este apartado lo dedicaremos a las funciones necesarias para el funcionamiento correcto en el último momento del algoritmo A* y las demás funciones de lanzamiento a nivel usuario.

Aquí acabamos de formar lo que sería la última estructura, la pila de nodos, con las operaciones típicas de las pilas, SacaPila (que saca un nodo), MetePila (en el que introduce un nodo) y VerPila que muestra el nodo que se encuentra en la cima. Para hacer la función de destructor, se encuentra implementada la función BorrLisNodo. La función constructora es CreaAbiertos, en la que también llamamos a la función para crear el nodo inicial.

```
      matriz->numelementos=matriz->numelementos+1;
      //abiertos=(nodo**)realloc(abiertos,(aux+1)*sizeof(nodo*));
      matriz->abiertos=(nodo**)realloc(matriz->abiertos,
sizeof(nodo*)*(matriz->numelementos+1));
      matriz->abiertos[matriz->numelementos]=NULL;
      matriz->abiertos[matriz->numelementos]=dato;
```

Función de MetePila

Luego nos encontramos dos funciones muy similares a dos mostradas anteriormente, la de VolcTable y la de VolcTable2D. Tienen la misma función, lo único que en esta vuelca el resultado en un fichero. Además muestran más parámetros, como el tiempo tardado, los nodos analizados, el tiempo por nodo o el número máximo de nodos en la pila.

Antes de empezar con las funciones más relevantes, analizamos la función que decidirá si una ficha puede o no puede colocarse en la posición deseada, Ajuste. Esta es, junto con el modelo de representar cada pieza, la función más intuitiva y representativa a todos los puzzles, la más común y que todos realizamos cuando hacemos manualmente un puzzle. Esencialmente lo que hace es comparar los bordes de la pieza elegida, y mira los bordes de las piezas colindantes. Además, para poder mejorar la rapidez, miramos que no pertenezca al borde y estemos intentando ponerla en la parte interior del puzzle, o que sea una esquina y la intentemos poner en la zona

media de un borde. Esto lo conseguimos con comparaciones, no hay otra forma. Devuelve un booleano, 0 si es falso y 1 si es verdadero.

```
	//Miramos si son bordes y están en la zona central
	if(elegida->borde>0){
		if(actual->target[0]!=1 && actual->target[0]!=N && actual-
>target[1]!=1 && actual->target[1]!=M)
			return 0;
	}
	//Borde inferior
	if(actual->target[0]==N){
		if(actual->tab[actual->target[0]+1][actual->target[1]]-
>posiciones[0]!=elegida->posiciones[2])
			return 0;
	}
```

Extracto de Ajuste

La siguiente función es la que la utiliza, NuevosSucesores. Es una función que devuelve una lista de todos aquellos nodos que son sucesores. Si no hay ningún sucesor devuelve una casilla que apunta a null. La última casilla será un puntero a null.

```
	for(i=0;actual->inutil[i]!=NULL;i++)
	{
		if (Ajuste(actual,actual->inutil[i])==1)
		{
			lista=(nodo**)realloc(lista,(j+1)*sizeof(nodo*));
			lista[j]=ClonNodo(actual);
			InsertaFicha(lista[j],actual->inutil[i]);
			j++;
		}
	}
	lista=(nodo**)realloc(lista,(j+1)*sizeof(nodo*));
	lista[j]=NULL;
```

Bucle principal de NuevosSucesores

Por último vamos a analizar la función que cambia según la heurística o la forma de implementar el A*, AddAbiertos. En esta función elimina el primer nodo de Abiertos, desde el cual hemos sacado los nuevos sucesores, y se añaden a la matriz. En la primera versión, en la que se coge el primero el mejor siempre, el bucle era bastante simple:

```
		ElimNodo(SacaPila(matriz));
		iter=0;
		while(sucesores[iter]!=NULL){
			MetePila(matriz,sucesores[iter]);
			iter++;
		}
		free(sucesores);
```

Primera versión

Pero en cuanto le obligamos a que sea aleatorio, el bucle se complica y hay que hacer nuevas funciones.

```
       ElimNodo(SacaPila(matriz));
       limiteInf=0;
       limiteSup=1;
       if(sucesores[limiteInf]!=NULL){
              while(sucesores[limiteSup]!=NULL){
                     if(sucesores[limiteSup]->ultheuristica !=
sucesores[limiteInf]->ultheuristica){
                            MeteRandom(matriz,sucesores,limiteSup,limiteInf);
                            limiteInf=limiteSup;
                     }
                     limiteSup++;
              }
              MeteRandom(matriz,sucesores,limiteSup,limiteInf);
       }
```

Versión posterior utilizada en las aleatorias

Este caso es la versión más avanzada de la heurística, en la que había que intentar colocar siempre las piezas por su heurística. Esto hace que para que fuera aleatorio, hace falta ir metiendo los nodos por rangos de heurística. Y claro, para ello necesitamos dos funciones más, una que depende de otra. Una, MeteRandom, introduce en la pila los sucesores que marcas entre los límites de forma aleatoria.

```
       ele=random(limiteSup-limiteInf);
       for(aux=ele+limiteInf;aux>=0&&aux>=limiteInf;aux--)
              MetePila(matriz,sucesores[aux]);
       for(aux=limiteSup-1;aux>=limiteInf+ele+1&&aux>=limiteInf;aux--)
              MetePila(matriz,sucesores[aux]);
```

Marca un nodo como elegido y primero mete los nodos que se encuentra por debajo de él y luego el resto

Y por último se encuentra la función random. Esta función al darle el número de nodos que se van a introducir, elige un número aleatoriamente que se encuentre en ese rango. Para que nunca coincida, cada vez que se le llama a la función se inicia otra vez la semilla de la función random propia de C respecto al tiempo y del número de nodos. Como estamos operando en Windows es necesario utilizar funciones propias de tiempo de Windows.

```
int random(int nsuc){
       LARGE_INTEGER t1;
       QueryPerformanceCounter(&t1);
       srand(t1.QuadPart*(nsuc+1));
       return (int)( (double)rand()/(double)(RAND_MAX+1)*nsuc);
}
```

3.5 LANZAMIENTO FINAL DEL ALGORITMO

Por fin llegamos al último nivel de implementación. En él lo que vamos a hacer es englobar las funciones codificadas en secciones anteriores y darles un sentido para que el algoritmo A* funcione correctamente.

Está compuesto por tres funciones importantes, y una más que es local que sirve para manejar el tiempo, que aquí en Windows es diferente y se necesitan medidas de alta precisión.

Empezamos con esta última, ya que el resto se dan sentido las unas a los otras y esta es la más aislada. En Linux, con meternos en la librería *"sys/time.h"* tenemos el problema resuelto, ya que hay una gran cantidad de formas de sacar el tiempo con una precisión bastante buena. Por ejemplo existe la función *clock()*, que con hallar la diferencia entre dos variables de tipo *clock_t* y dividirlo por la constante correspondiente tienes los segundos. Pero en Windows esto tiene un pequeño problema, que es que no tiene una precisión ni de 10 milisegundos, lo cual implica que por ejemplo para nuestro programa no sirve ya que mide tiempos muy reducidos. Otra posible función que en Linux funciona bien *gettimeofday*, utilizando la estructura *timeval*, pero volvemos a tener el mismo problema. Para solucionarlo hay que utilizar lo que se suele denominar contador de alta resolución. La función en cuestión es parecida a sus similares que hemos explicado anteriormente, pasando dos estructuras especiales y dividiendo por una constante:

```
double performancecounter_diff(LARGE_INTEGER *a, LARGE_INTEGER *b)
{
	LARGE_INTEGER freq;
	QueryPerformanceFrequency(&freq);
	return (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
}
```

Luego para utilizarla solo tienes que hacer este tipo de razonamiento.

```
LARGE_INTEGER t_ini, t_fin;
double secs;
QueryPerformanceCounter(&t_ini);
/* ...hacer algo... */
QueryPerformanceCounter(&t_fin);
secs = performancecounter_diff(&t_fin, &t_ini);
```

Primeramente analizamos la función Inicio. Esta función engloba todos los procesos necesarios para poder iniciar el algoritmo, creando Abiertos y metiendo todas las fichas en la tabla TodFichas una vez que se han sacado de los ficheros. Tiene tres opciones, según sea la situación en la que te encuentres. La primera sirve cuando

tienes que transformar el fichero original de las 256 piezas. La segunda opción es la más utilizada, que es cuando tienes que cogerla de un fichero ya transformado, mientras que la última se utiliza cuando el fichero aún no tiene introducidas las fichas generadas por los giros.

Ahora vamos a analizar la función Aest, la función A*. Esta función es la que verdaderamente mueve todo el funcionamiento del programa.

```
       while(VerPila(matriz)!=NULL && EsFinal(VerPila(matriz))==0 )
       {
              AddAbiertos(matriz,NuevosSucesores(matriz->abiertos[matriz-
>numelementos]));
                     .
                     .
       }
       if(VerPila(matriz)!=NULL)
              return VerPila(matriz);
```

Estructura principal del Aest

La primera línea es la comparación del bucle, que va de izquierda a derecha. Primeramente revisa que el elemento que se encuentra en la cima de la pila exista. Posteriormente se mira si el nodo es el que buscamos. En algunas versiones hay una tercera cláusula que sirve para limitar el número de nodos o el tiempo que tarda. La segunda línea es la que hace el resto del A estrella. En ella primeramente sacamos los nuevos sucesores de la cima, para posteriormente llamar a la función AddAbiertos y que elimine el nodo y los añada. El resto del bucle son instrucciones de control del programa, que muestra periódicamente el tiempo que ha pasado o cuantos nodos se han analizado. Finalmente revisa si es final o no y lo devuelve.

Vamos a analizar ahora el bucle principal. Para poder hacer varias medidas seguidas sin tener que lanzar una y otra vez el programa, todo esta englobado dentro de un for. Primeramente, reservamos memoria para la pila, después llamamos a la función para sacar las fichas y crear toda la situación inicial, empezamos a tomar la medida del tiempo y lanzamos el A*.

Posteriormente, esta un bloque en el que va encauzando el resultado a diferentes ficheros. Por último, liberamos toda la memoria utilizada durante el programa. El último *Sleep* sirve para las simulaciones de los puzzles pequeños, al ser tan pequeño el tiempo utilizado, interiormente el *random* no utiliza una nueva semilla, y al hacerlo dormir le obligamos a ello para que salgan simulaciones diferentes.

4 RESULTADOS Y COMPARACION

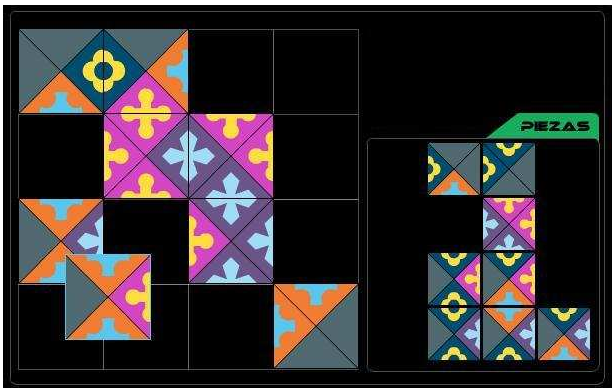
4.1 CONDICIONES DE LAS PRUEBAS

Para que las medidas tuviesen la misma validez, todas las pruebas fueron realizadas en el mismo ordenador, y sin programas adicionales ejecutándose a la vez. Este ordenador es un Intel Core Duo a 1.97 Ghz y 3 Gb de RAM.

4.2 RESULTADOS

4.2.1 1º TEST

El primer test lo realizamos sobre la prueba online que hay en la pagina oficial del puzzle. En esta prueba nos muestran un tablero de 4x4 con 16 fichas, y cuatro colores diferentes (contando los bordes).



Este primer test esta resuelto de la primera forma que comentamos en la sección de *CASO CONCRETO*, eligiendo el primero el mejor siempre sin heurística y sin ser aleatorio.

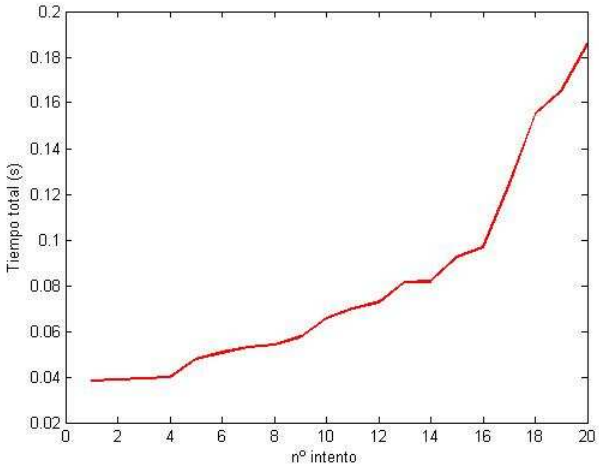
Tiempo tardado: 0.062322
Nodos Analizados: 43
Tiempo por nodo: 0.001449
Nodos Abiertos: 21

4.2.2 2º TEST

El segundo test lo realizamos sobre las mismas condiciones que el anterior, sobre el tablero de 16 piezas. Esta vez también trabajaremos sin heurística, pero esta vez cogeremos las piezas de forma aleatoria. Para comparar y hacer un estudio más estadístico, lanzaremos el programa 20 veces.

*Tiempo Total

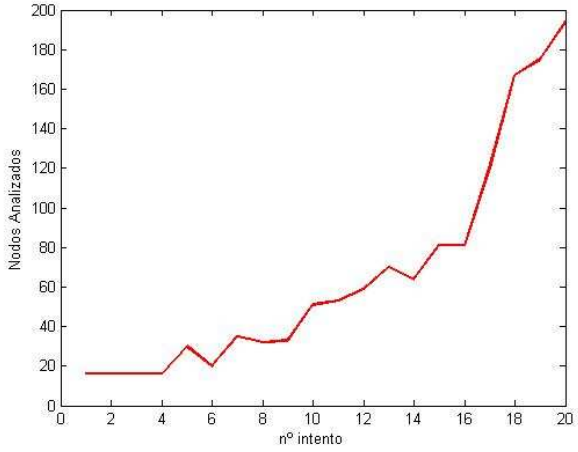
En esta primera gráfica podemos observar el tiempo total en segundos, utilizado por el programa para resolver el puzzle. Esta ordenado de menor a mayor para ver mejor el rango en el que se mueven las medidas.



Tiempo Mínimo: 0.038230
Tiempo Máximo: 0.185778
Tiempo Medio: 0.080509

*Nodos Analizados

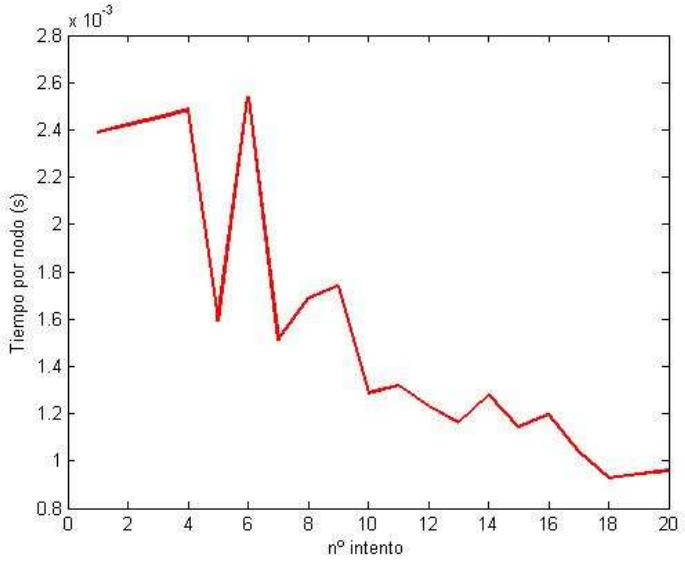
En la segunda gráfica podemos observar el número de nodos analizados en cada una de las simulaciones. Esta y el resto de gráficas serán ordenadas según el tiempo total de ejecución.



Nodos Mínimos: 16
Nodos Máximos: 194
Nodos Medios: 66.45

*Tiempo por Nodo

La tercera gráfica sirve para calcular el tiempo medio para calcular un nodo:



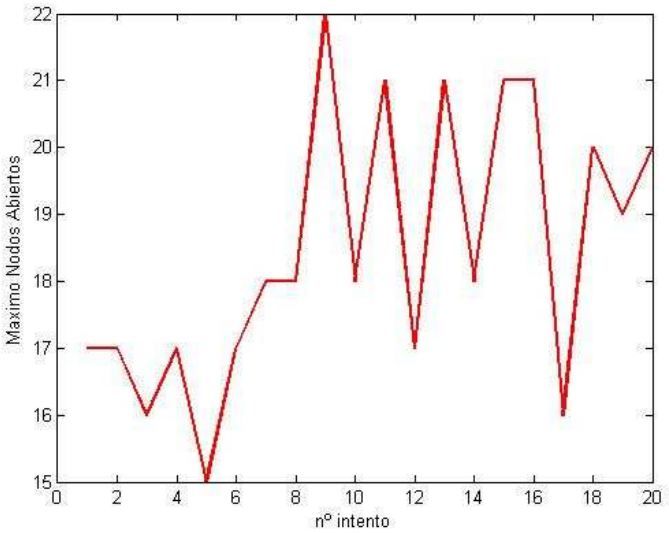
Tiempo Mínimo: 9.28 x 10^{-4}

Tiempo Máximo: 0.002539

Tiempo Medio: 0.001565

*Numero Máximo de Nodos En Abiertos

La última gráfica nos muestra el número de nodos abiertos en el momento de mayor expansión:



Número Mínimo de nodos: 15

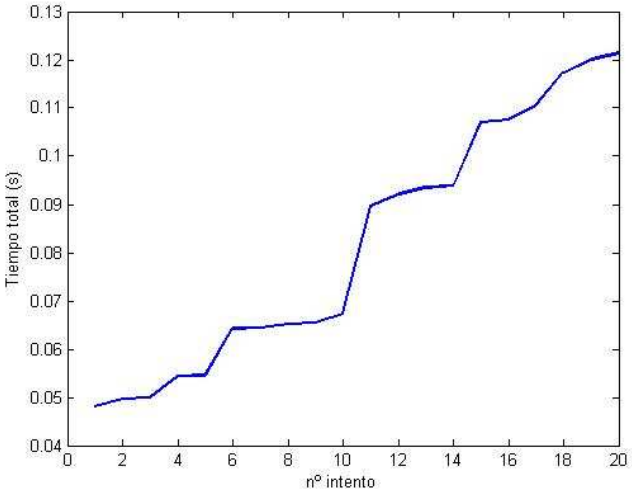
Número Máximo de nodos: 22

Número Medio de nodos: 18.45

4.2.3 3º TEST

El tercer test lo realizamos sobre las mismas condiciones que los dos anteriores, sobre el tablero de 16 piezas. Esta vez sí utilizaremos la heurística mencionada en el *CASO CONCRETO*, y también cogeremos las piezas al azar. Para comparar y hacer un estudio más estadístico, lanzaremos el programa 20 veces.

***Tiempo Total**

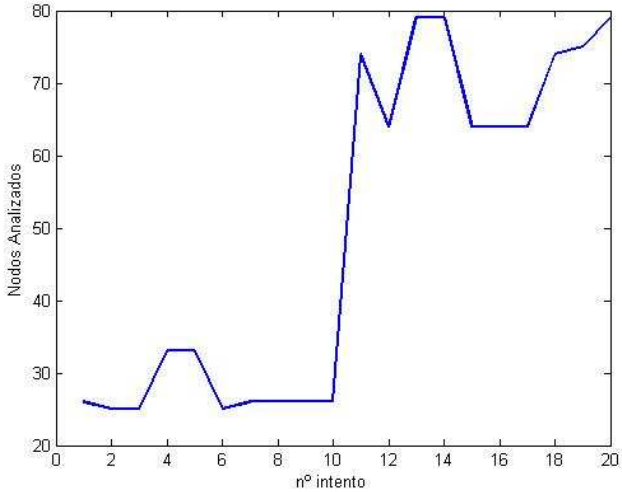


Tiempo Mínimo: 0.048045

Tiempo Máximo: 0.121434

Tiempo Medio: 0.081832

***Nodos Analizados**

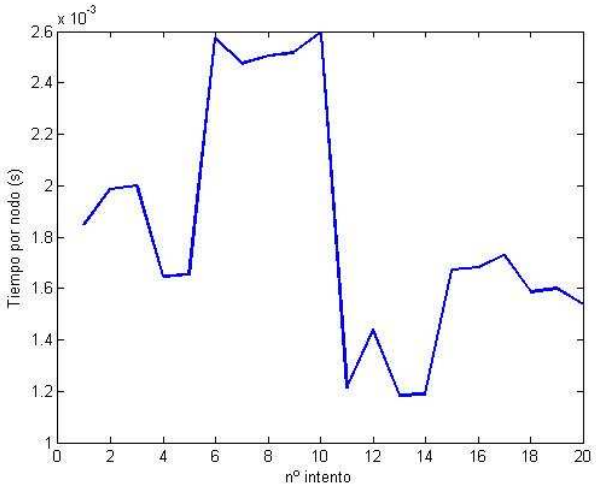


Nodos Mínimos: 25

Nodos Máximos: 79

Nodos Medios: 49.35

**Tiempo por Nodo*

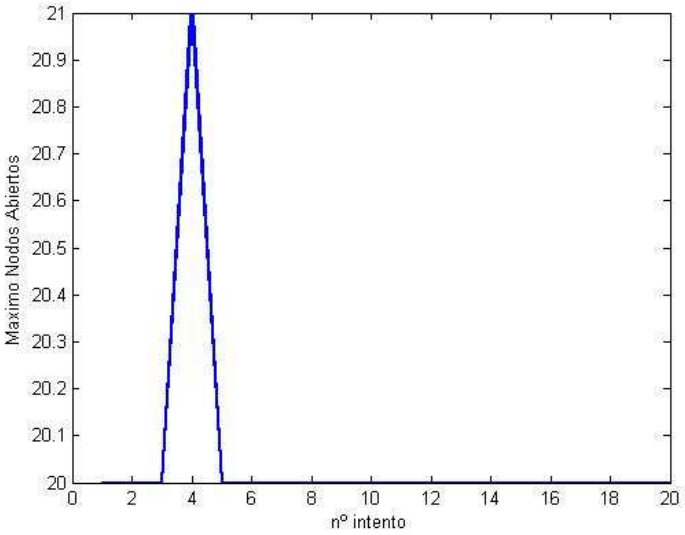


Tiempo Mínimo: 0.001184

Tiempo Máximo: 0.002595

Tiempo Medio: 0.001836

**Numero Máximo de Nodos En Abiertos*



Número Mínimo de nodos: 20

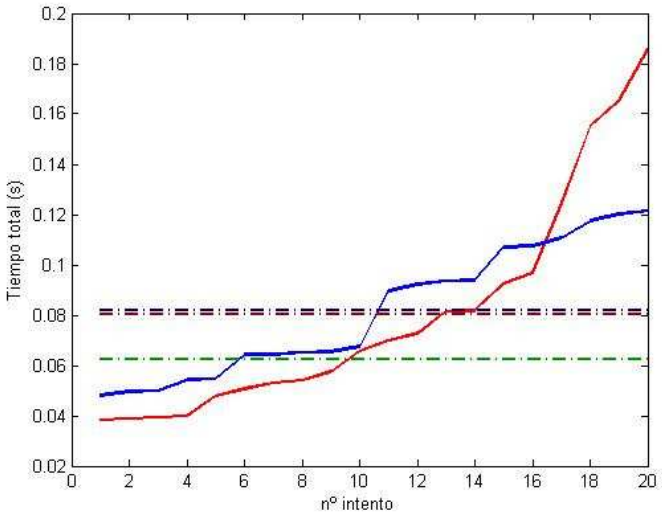
Número Máximo de nodos: 21

Número Medio de nodos: 20.05

4.2.4 1ª COMPARACION

La comparación se da entre los tres test chequeados anteriormente.

Tiempo Total



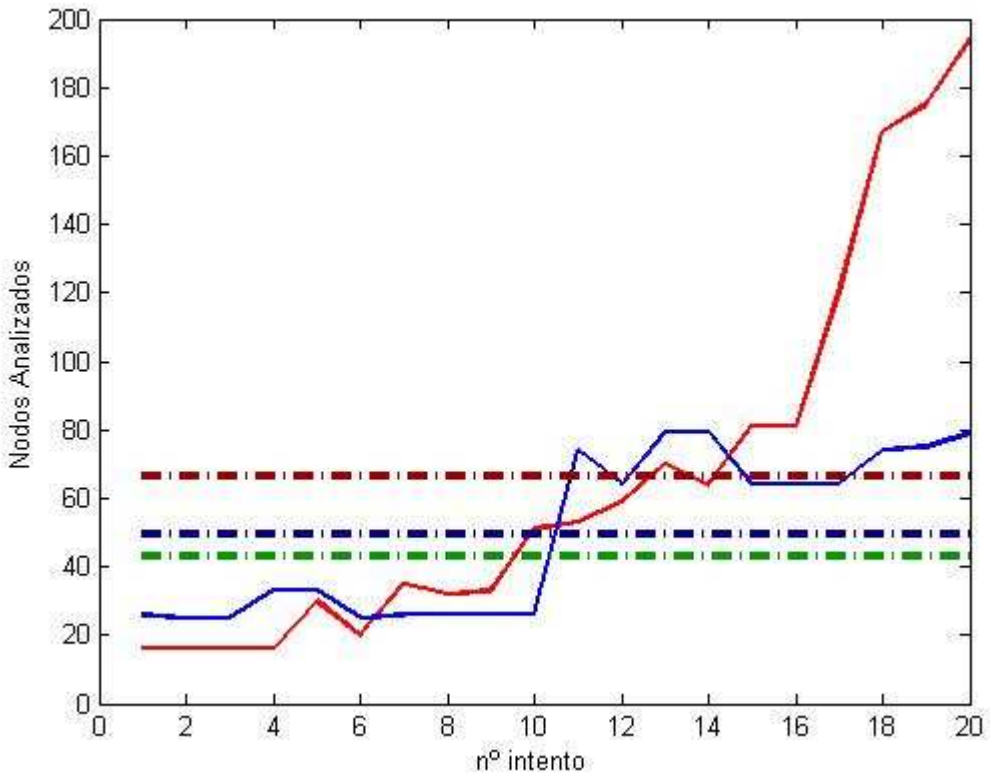
	PRIMERO	RANDOM	HEURISTICA
Mínimo	0.062322	0.038230	0.048045
Máximo	0.062322	0.185778	0.121434
Media	0.062322	0.080509	0.081832

Respeta los colores que hemos utilizado en las gráficas anteriores, siendo rojo el segundo test (RANDOM), azul el que utiliza heurística y verde el primero el mejor. Además las líneas punteadas nos muestran las medias.

En esta gráfica podemos observar como se comportan los tres modelos. Por ejemplo, podemos observar que el sistema de primero el mejor es el que mejor funciona en este caso, pero hay que tener en cuenta que depende mucho de cómo se encuentren las fichas ordenadas, y en el peor de los casos puede dar un tiempo muy superior a este y ser mucho peor.

También observamos que las medias son casi iguales, lo que hace suponer que tienen gráficas similares, pero fijándonos más se observa que lo que hace que aumente tanto el valor total son una serie de valores aislados, mientras que la heurística es mucho más constante pero globalmente superior al segundo test. Por estadística tenemos mejor rendimiento con el segundo, ya que excepto en un 20 % de los casos, el tiempo es claramente inferior.

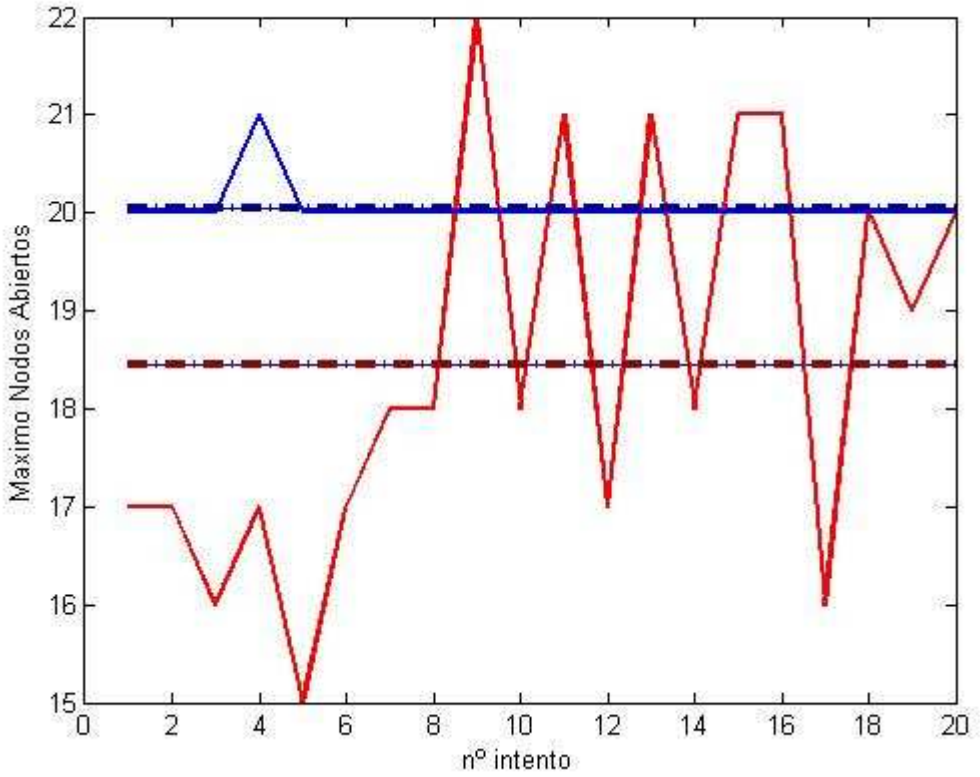
**Nodos Analizados*



	PRIMERO	RANDOM	HEURISTICA
Mínimo	43	16	25
Máximo	43	194	79
Media	43	66.45	49.35

En este caso se ven mayores diferencias. Podemos observar como de media se analizan 20 nodos más si se simulan sin heurística que de cualquiera de las otras formas. Volvemos a observar que también es el que más fluctúa, con gran diferencia entre el máximo y el mínimo. Vuelve a pasar lo mismo, en el 20 % de las simulaciones el aleatorio da un rendimiento muy inferior a los demás. El de heurística tiene una gráfica en cambio muy constante, con un rango de valores mucho más compacto. El primero el mejor sigue siendo mejor en este caso.

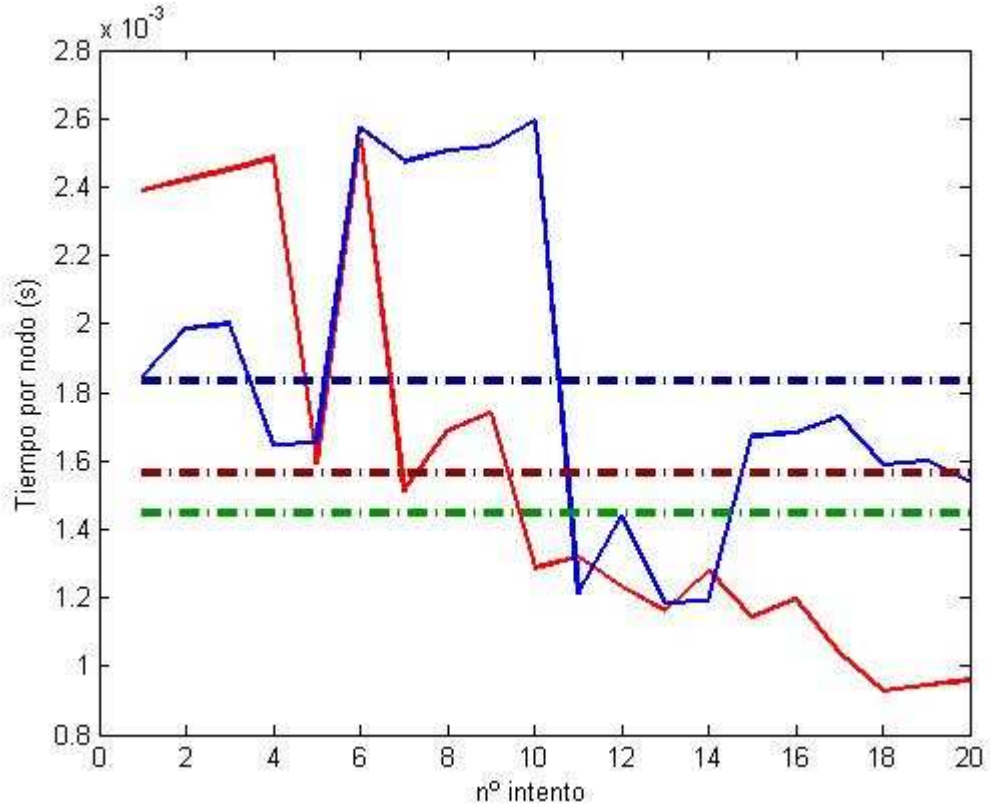
**Numero Máximo de Nodos En Abiertos*



	PRIMERO	RANDOM	HEURISTICA
Mínimo	21	15	20
Máximo	21	22	21
Media	21	18.45	20.05

Esta es la única medida en el que el primer test resulta peor que los demás, a pesar de ser todos muy parecidos.

**Tiempo por Nodo*



	PRIMERO	RANDOM	HEURISTICA
Mínimo	0.001449	9.28×10^{-4}	0.001184
Máximo	0.001449	0.002539	0.002595
Media	0.001449	0.001565	0.001836

En esta gráfica apreciamos que el tiempo por nodo del test con heurística es superior a los otros. También observamos que en el segundo test, según se analizan más nodos y pasa más tiempo, se disminuye el tiempo por nodo.

4.2.5 4º TEST

El cuarto test lo realizamos sobre el primer puzzle pista que nos proporciona Eternity. En esta prueba nos muestran un tablero de 6x6 con 36 fichas, y 8 colores diferentes.

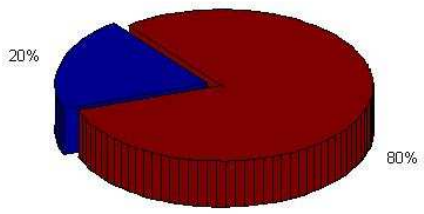
Esta resuelto de la primera forma que comentamos en la sección de *CASO CONCRETO*, eligiendo el primero el mejor siempre sin heurística y sin ser aleatorio.

Tiempo tardado: 1.688413
Nodos Analizados: 1228
Tiempo por nodo: 0.001375
Nodos Abiertos: 103

4.2.6 5º TEST

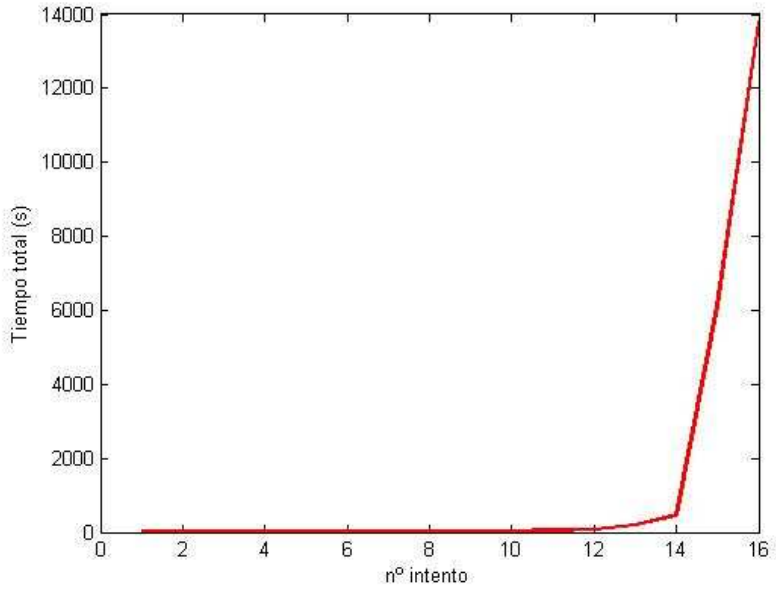
El quinto test lo realizamos sobre las mismas condiciones que el anterior, sobre el tablero de 36 piezas. El caso que simularemos será el que se cogen las piezas al azar sin tener en cuenta ninguna heurística, el segundo supuesto mencionado en el *CASO CONCRETO*. Para comparar y hacer un estudio más estadístico, lanzaremos el programa 20 veces. Debido al tiempo necesario que requieren algunas simulaciones, limitaremos el programa a 18 millones de nodos, 6 horas de ejecución, ya que si lo dejamos simulando hay algunos que tardan varios días. Los que superen esto los denominaremos fallos.

Tasa de Fallos:



En el estudio hubo un 20 % de las simulaciones no validas. Por lo que el resto de graficas serán sobre 16 simulaciones.

**Tiempo Total*

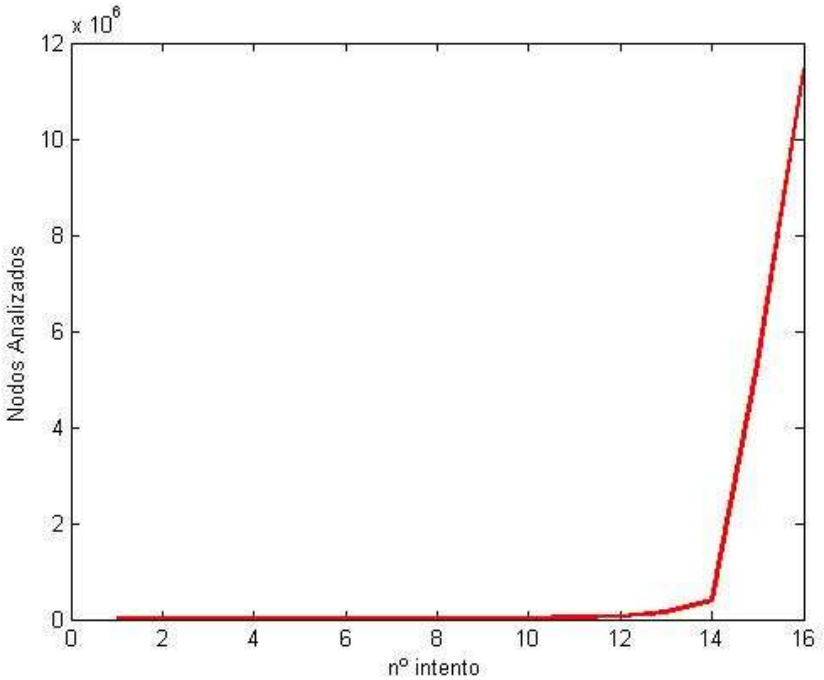


Tiempo Mínimo: 1.38427

Tiempo Máximo: 1.383277051 x 10^4

Tiempo Medio: 1.2998012 x 10^3

**Nodos Analizados*

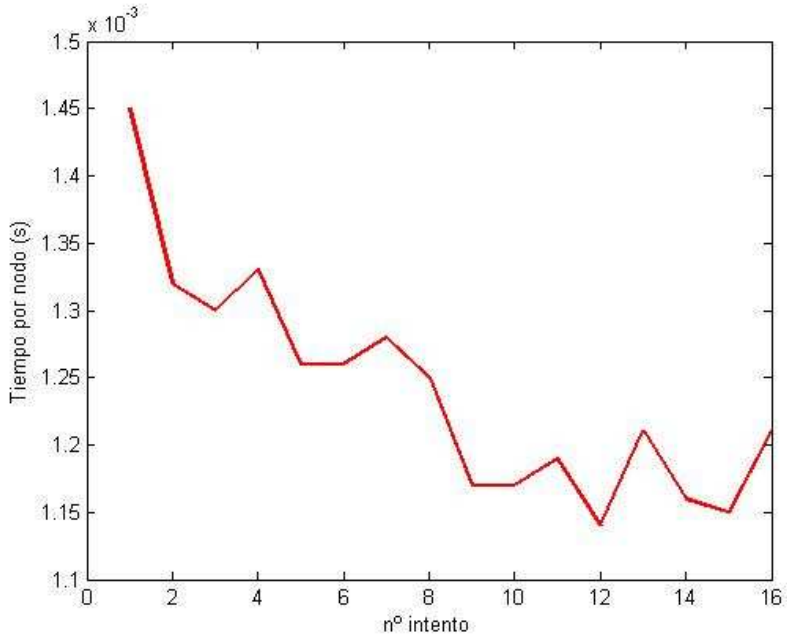


Nodos Mínimos: 957

Nodos Máximos: 11467388

Nodos Medios: 1.0947915 x 10^6

**Tiempo por Nodo*

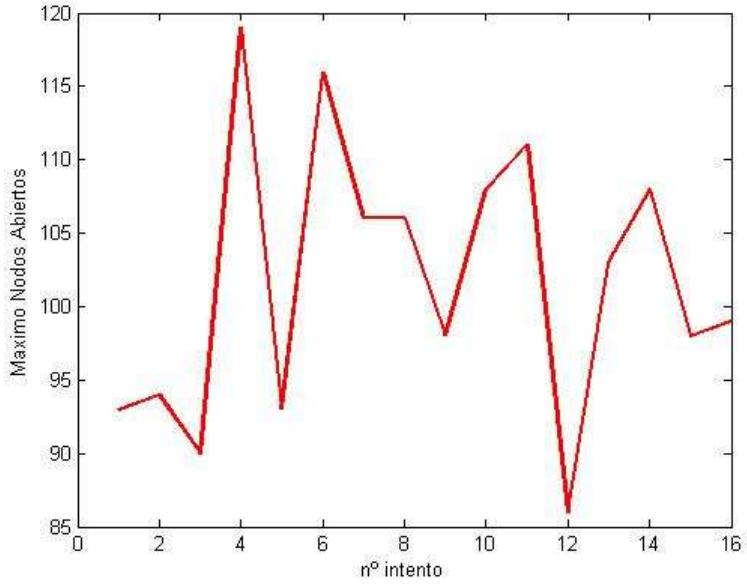


Tiempo Mínimo: 0.00114

Tiempo Máximo: 0.00145

Tiempo Medio: 0.001240625

**Numero Máximo de Nodos En Abiertos*



Número Mínimo de nodos: 86

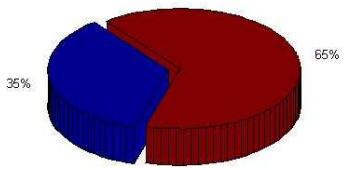
Número Máximo de nodos: 119

Número Medio de nodos: 101.75

4.2.7 6º TEST

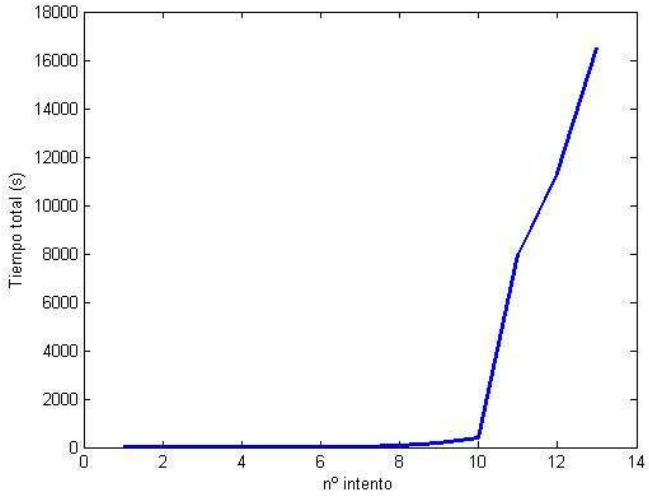
El sexto test lo realizamos sobre las mismas condiciones que los dos anteriores, sobre el tablero de 36 piezas. Volveremos a utilizar la heurística descrita en *CASO CONCRETO*, cogiendo las piezas al azar. Para comparar y hacer un estudio más estadístico, lanzaremos el programa 20 veces. También, como en el caso anterior, limitaremos el programa a 18 millones de nodos, 6 horas de ejecución, los que tarden más pasaran a formar parte de la tasa de fallos.

Tasa de Fallos:



Nos encontramos con una tasa de fallos del 35%

Tiempo Total

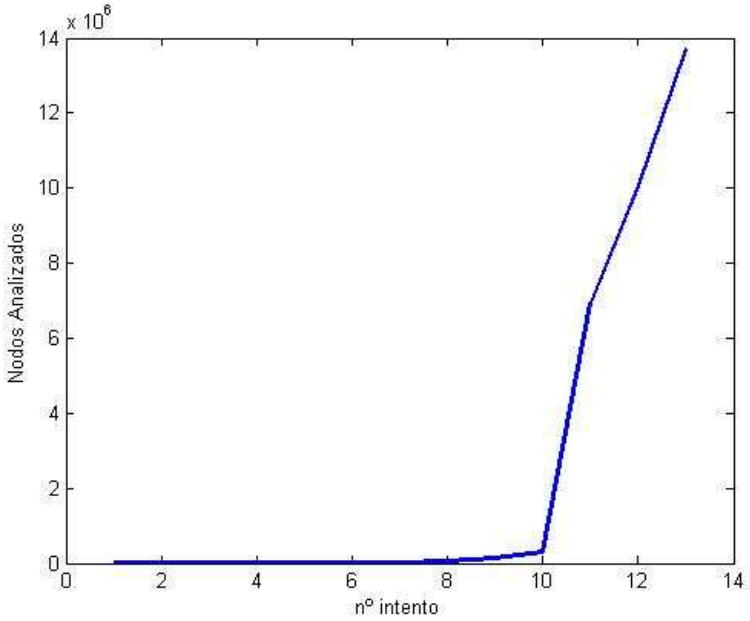


Tiempo Mínimo: 2.74154

Tiempo Máximo: 1.6501025 x 10^4

Tiempo Medio: 2810.0389

**Nodos Analizados*

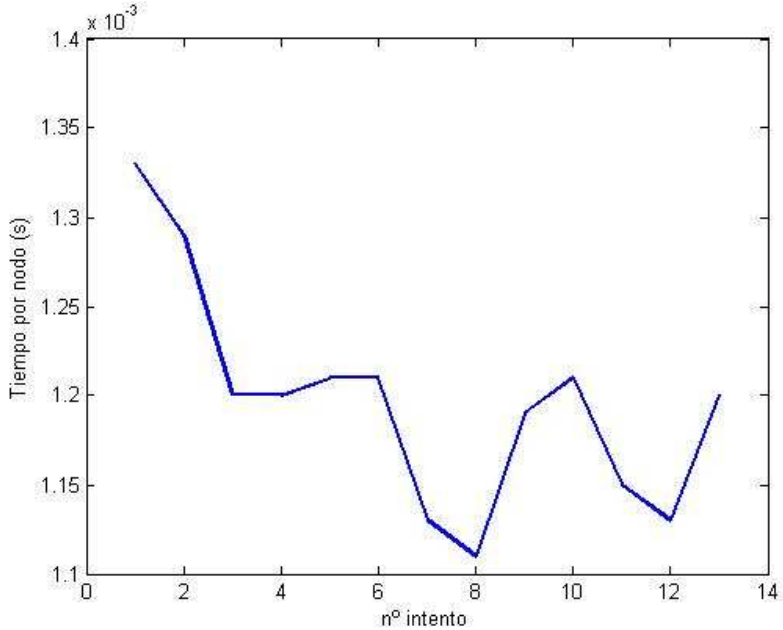


Nodos Mínimos: 2067

Nodos Máximos: 13705669

Nodos Medios: 2.4071655 x 10^6

**Tiempo por Nodo*

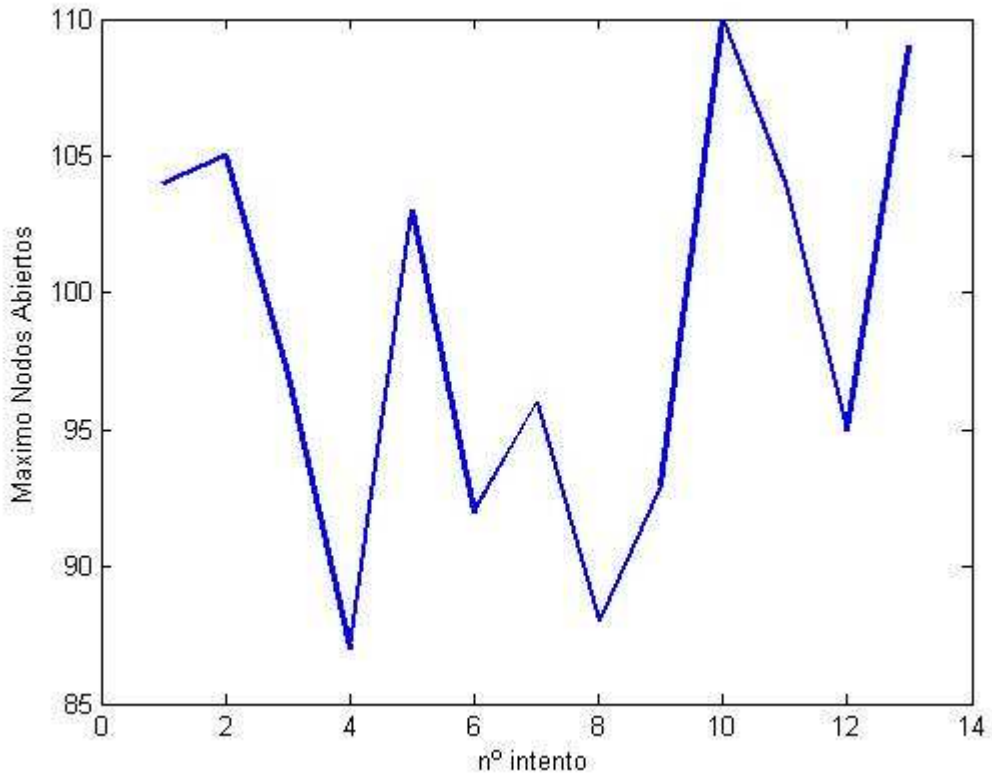


Tiempo Mínimo: 0.00111

Tiempo Máximo: 0.00133

Tiempo Medio: 0.001196

**Numero Máximo de Nodos En Abiertos*



Número Mínimo de nodos: 87

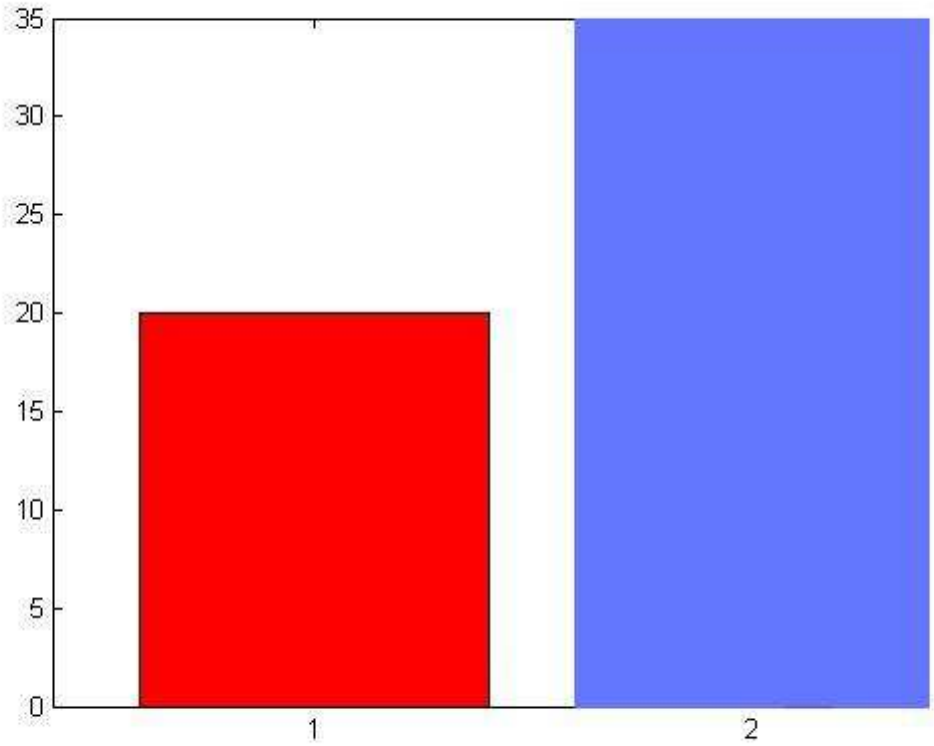
Número Máximo de nodos: 110

Número Medio de nodos: 98.69

4.2.8 2ª COMPARACION

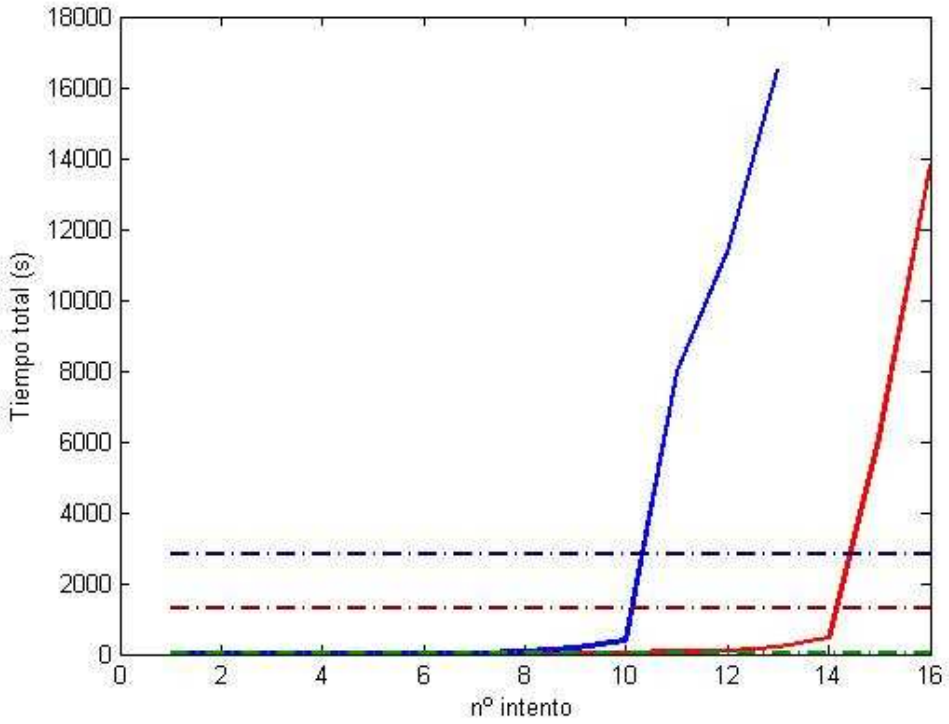
Esta comparación se hará sobre los test 4, 5 y 6. Los campos que se verán son la tasa de fallos, tiempo total, número de nodos, tiempo por nodo y número de nodos máximo en abiertos.

Tasa de Fallos:



El color rojo se corresponde al test que no tiene heurística, el test 5, mientras que el azul es el que tiene heurística, el test 6. Observamos que la tasa de fallos del azul (un 35 %) es casi el doble que la roja (un 20 %).

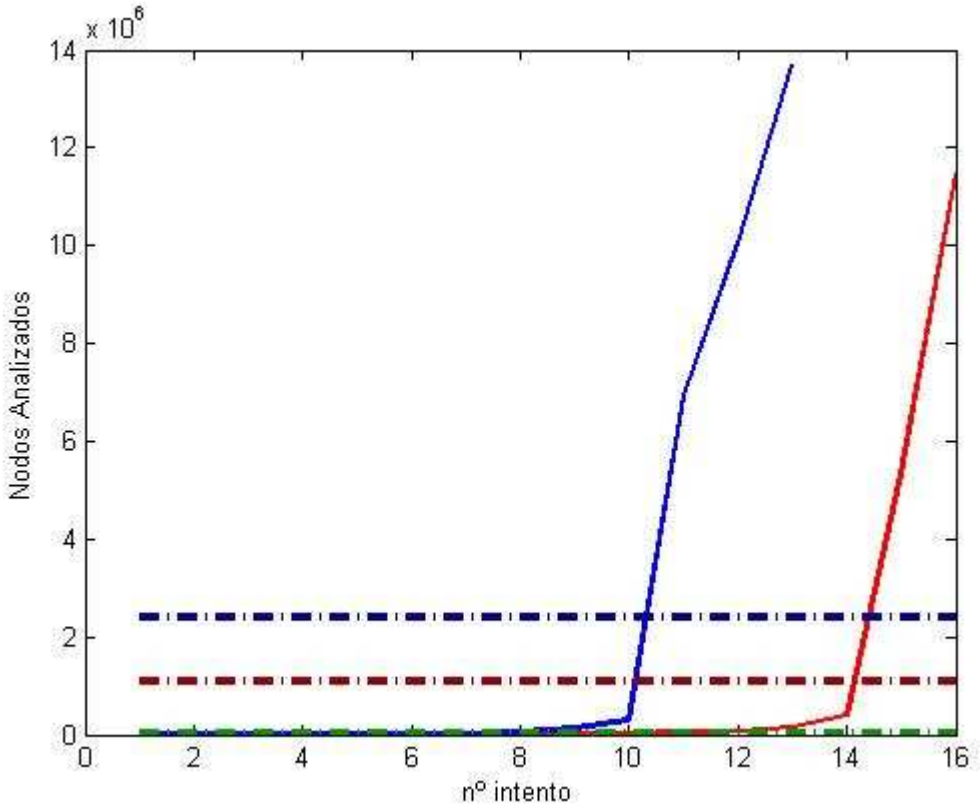
**Tiempo Total*



	PRIMERO	RANDOM	HEURISTICA
Mínimo	1.688413	1.38427	2.74154
Máximo	1.688413	1.383277×10^4	1.650103×10^4
Media	1.688413	1.299801×10^3	2.810039×10^3

Se observa en las dos gráficas que hay dos tipos de simulación, en una de ellas, el tiempo que se tarda es muy reducido, alcanzando el camino correcto en pocas iteraciones, mientras que el segundo tipo el tiempo de simulación se dispara. Lo que hay que tener en cuenta es que el primer tipo de simulaciones es menor en heurística que en el de random, por ello la media de tiempo es mucho mayor (el doble). También el resto de medidas son superiores en heurística que en el de random. Si comparamos el random y el de heurística con el primero el mejor, vemos que solo algún valor del random puede mejorarlo, pero es mucho mejor por lo general, aunque ya explicamos como funcionaba en este caso este algoritmo, que depende mucho de cómo estén las fichas ordenadas.

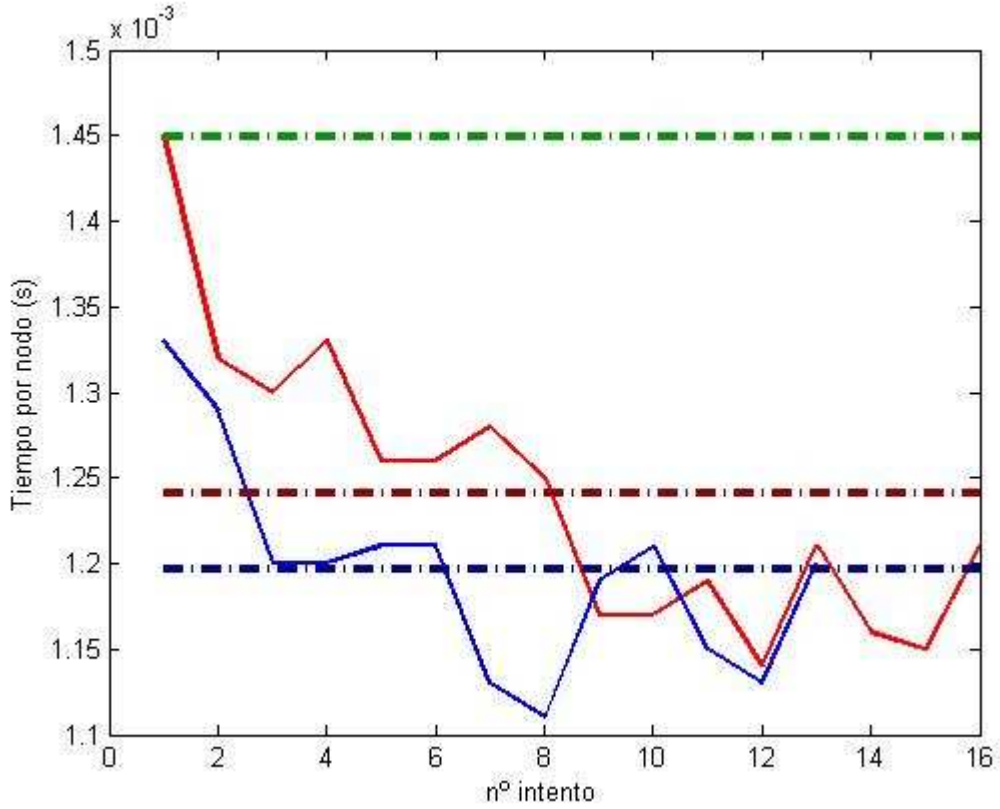
**Nodos Analizados*



	PRIMERO	RANDOM	HEURISTICA
Mínimo	1228	957	2067
Máximo	1228	11467388	13705669
Media	1228	1094791.5	2407165.5

Esta gráfica es similar a la anterior, el primero el mejor es sólo superado por algún caso aislado de random, pero por el resto random tiene la misma gráfica que el de heurística, con valores finales cercanos al umbral de fallo. Como en el caso anterior, el grupo de datos que no están saturados es bastante menor en el de heurística que en el de random, lo que implica que la media sea superior.

**Tiempo por Nodo*

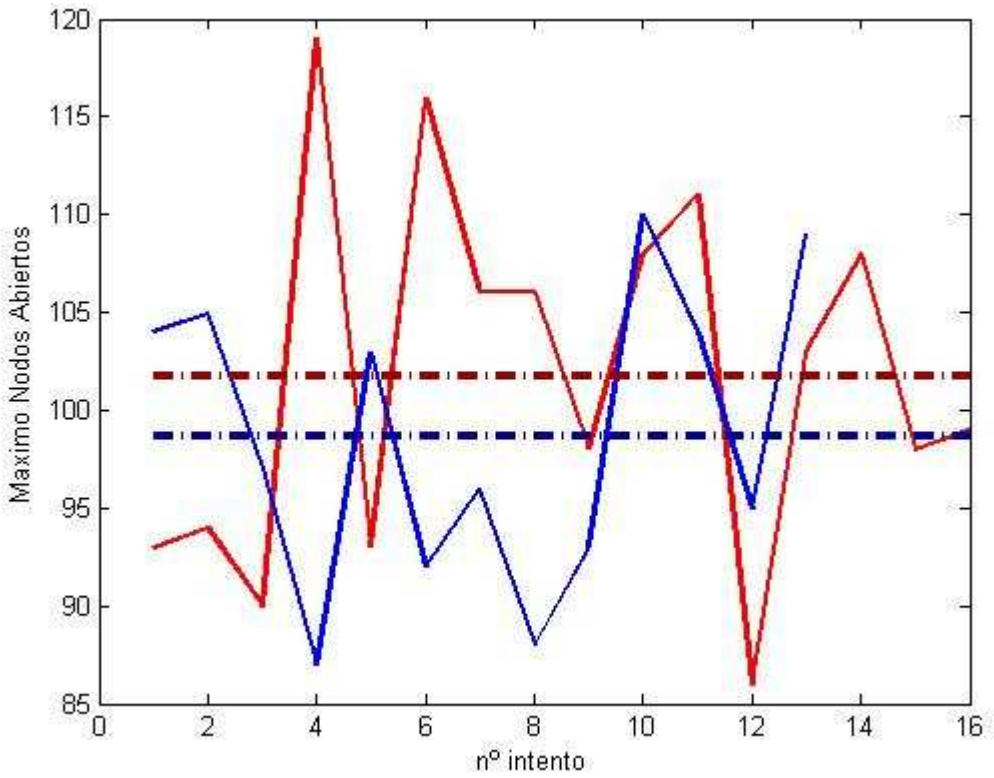


-

	PRIMERO	RANDOM	HEURISTICA
Mínimo	0.001375	0.001140	0.001110
Máximo	0.001375	0.001450	0.001330
Media	0.001375	0.001240	0.001196

Esta gráfica nos vuelve a mostrar que a mayor tiempo que pase, mayor rendimiento tiene el ordenador al analizar los nodos. Por ello vemos que heurística, la gráfica que mayor tiempo tardaba, es la que menor tiempo por nodo utiliza de media, mínimo y máximo, y el de primero el mejor, que es la más recomendada según las anteriores estimaciones, es la peor. La diferencia entre las medias de random y heurística no es muy pronunciada, pero en simulaciones de gran cantidad de nodos puede ser significativa.

**Numero Máximo de Nodos En Abiertos*



	PRIMERO	RANDOM	HEURISTICA
Mínimo	103	86	87
Máximo	103	119	110
Media	103	101.75	98.69

En esta gráfica no hay mucho reseñable, salvo que los valores de la función heurística están más cercanos a su media que los de random, y sus valores son los menores de los tres, aunque los tres test tienen medias similares.

5 CONCLUSIONES, PROBLEMAS Y TRABAJOS FUTUROS

5. 1 CONCLUSIONES

Como hemos podido observar, el mejor método en estos casos ha sido el primero el mejor, aunque hay que tener en cuenta lo ya dicho anteriormente, que ha sido el mejor en este caso por la ordenación, pero también puede ser de los peores. También nos podemos fijar en que la heurística escogida en este caso ha sido un verdadero fiasco, no mejorando las estadísticas de cogerlo al azar. Pero también vemos que es la más constante y todas sus gráficas, lo que nos demuestra que el verdadero reto será elegir una heurística buena. También se ve que en cuanto el puzzle se hace más grande, por fichas o por el tamaño del tablero, el tiempo de ejecución se dispara, por lo que la ejecución del puzzle grande con la técnica y heurística anteriormente utilizada es totalmente inviable.

5.2 PROBLEMAS

En este proyecto, para que salga adelante, tienes que tener muy claro la metodología del A*, porque lo necesitas plasmar posteriormente en cualquier otro lenguaje. Por ejemplo, tal y como se implementa Abiertos, lo más fácil es implementarlo con forma de una pila, ya que te olvidas así de muchos líos posteriores.

Y luego es haberlo implementado en C. Se podría haber hecho en Java, por ser orientado a objetos, o cualquier similar, que al leerlo es más fácil de visualizar y ver en qué formas está dividido el programa, pero todos estos lenguajes suelen ser interpretados, lo que significa mucha menor eficiencia en tiempo y en memoria. En cambio C operas directamente con la memoria y lo puedes optimizar todo lo que necesites, además de que por las características de C, por ser compilado, es más rápido y eficiente con el almacenamiento de la memoria. Todo lo que tiene de bueno, lo tiene de malo si no lo haces con un orden, me explico:

En mi caso, como hemos visto en la sección de implementación, en el modelo, tenemos cuatro estructuras de datos, ficha, tablero, nodo y matriz, y cada una de estas estructuras forma parte de la estructura de datos superior a ella. Todo esto se maneja con punteros, punteros dobles… etc. Y si hacemos cálculos, por ejemplo en el caso

que queramos ver el color de una ficha desde el nivel de matriz, tienes que sumergirte en 9 niveles de profundidad. Y no solo eso, sino que hablando de puzzles grandes, la ejecución tarda varios días o semanas o más, y ya no se habla por cientos de nodos, se habla de miles de millones de nodos. Eso implica que tienes que liberar absolutamente todos los punteros, no hay la posibilidad de dejarte ni uno, y en un programa que son todo punteros, es complicado. Y eso es lo que ha pasado en este proyecto.

Después de cuatro meses implementando todo el código, llega el tiempo de probarlo por primera vez en conjunto, porque hasta entonces solo has podido hacer pruebas muy limitadas. Claro, la primera prueba que haces la haces sobre el online, que en el peor de los casos analiza medio centenar de nodos. Ahí no notas nada. Pero cuando empiezas a intentar resolver el puzzle grande, de 256 piezas, te da fallo de segmentación a las 10 horas de ejecución, sobre 23 millones de nodos aproximadamente. Entonces te empiezas a alarmar, viendo que el programa no es perfecto, que hay en algún sitio un detalle que te as dejado. Después de repasarlo unas cuantas veces, y de cambiar varias cosas en las que crees puede estar el fallo, te das cuenta que buscar así es como buscar una aguja en un pajar. Tras esto haces cálculos mirando la capacidad de memoria del ordenador, el tamaño de los punteros, y te das cuenta que por cada nodo que se analiza, pierdes un solo puntero, cuatro bytes. Ahí es cuando empiezas a hacer una y otra vez depuraciones exhaustivas del código, mirando las posiciones físicas de la memoria, pero al final te sabes tan bien el código que puedes ir haciéndolo casi sin necesidad de pantalla.

Finalmente descubres la existencia de herramientas para depurar la memoria, tipo Valgrind y similares. El problema es la plataforma donde has hecho todo el proyecto. C esta orientado hacia Linux, y todas estas herramientas están exclusivamente en este SO. Finalmente encuentras una y ves que comprender los resultados que te dan es un trabajo de locos, pero cuando ya no te queda otra, lo tienes que comprender si o si. Finalmente encuentras el fallo, una reserva de memoria antes de llamar a una función que vuelve a reservar la misma memoria.

Todo este proceso, en mi caso, ha durado 6 meses, desde Febrero hasta finales de Julio. Y seguramente este problema lo tendrá toda la gente que implemente proyectos con punteros a gran escala.

5.3 TRABAJOS FUTUROS

El desarrollo posterior de este proyecto puede coger varios caminos.

En el caso que quisiésemos generalizar el programa para resolver cualquier puzzle, lo único que debemos hacer es modificar el campo de la ficha que antes almacenaba los cuatro colores, y por cada uno de los colores colocar un vector con todos los colores que tiene un lado de una ficha. Posteriormente remodelaríamos la función ajuste para que compare vectores, y no solo un color, y con ello ya podemos resolver por fuerza bruta cualquier puzzle.

En el caso que quisiésemos seguir investigando en el Eternity, podemos empezar haciendo estudios estadísticos. Por ejemplo estimar cuanto tiempo es necesario para resolver el puzzle grande de 256 piezas. Esto se consigue fácilmente viendo cuanto tarda el programa en resolver puzzles 6x6, 7x7, 8x8… con las 256 piezas, y hacer una función y una gráfica. Esta era una de las partes que fue suprimida de este proyecto original por falta de tiempo debido al problema anteriormente descrito, ya que estas pruebas tardan varios días (por ejemplo, el de 6x6 lo tuve que parar a 8 días de ejecutarse, y posiblemente hubiese tardado bastante más).

Si se tienen más ordenadores, se puede hacer que el tiempo de ejecución se divida proporcionalmente de forma realmente fácil. Por ejemplo, sea como sea la colocación de las piezas, hay cuatro piezas que son invariables y están en la misma posición, las esquinas. Si disponemos de tres ordenadores, podemos ejecutar el programa a la vez y cambiando el tablero del nodo inicial, fijar la esquina superior izquierda con una de las esquinas, siempre la misma, y la otra esquina superior irla variando en cada uno de los ordenadores. Esto hace que el árbol de decisión que en un principio tenemos para un ordenador, se divida en tres ramas y cada ordenador sólo se encarga de una de ellas. Teóricamente el puzzle original tiene una única solución. Esta forma de trabajo se puede escalar hasta 6 ordenadores sólo fijando las esquinas.

Otra de las cosas que hay que cambiar es la heurística. Se puede con esta misma heurística probar a ponerlas al revés, o cambiarla totalmente. Si nos fijamos en cómo resolvieron el Eternity 1, primero vieron cuales eran las piezas que se colocaban inicialmente si se aplicaba el algoritmo de forma aleatoria. Después de hacer un gran número de pruebas, analizaron los resultados y asignaron a cada ficha una heurística según el número de veces que se utilizaba. En la posterior ejecución del programa, siempre se intentan poner primero las que menos se utilizan estadísticamente, y así consiguieron resolverlo. En nuestro caso, se puede hacer lo mismo exactamente, es

más, era otra de las cosas que tuvimos que suprimir por falta de tiempo. Sobre un tablero de 10 x 10, con todas las fichas, intentar resolverlo un número concreto de veces y hacer una estadística de cada pieza, para ver cuantas veces se coloca o no.

6 BIBLIOGRAFÍA

HISTORIA DE PUZZLES

http://www.puzzletron.info/historia_puzzles.html

http://www.presentpics.com/tabid/58/articleType/ArticleView/articleId/4/Descubre-la-historia-del-Puzzle.aspx

http://www.woodentoys-uk.co.uk/jigsaw-history.html

http://www.mgcpuzzles.com/mgcpuzzles/puzzle_history/

HISTORIA DE LA INTELIGENCIA ARTIFICIAL

http://www.angelfire.com/falcon/miqueleiz/04HistoriadelaIA.htm

http://www.monografias.com/trabajos16/inteligencia-artificial-historia/inteligencia-artificial-historia.shtml

http://es.wikipedia.org/wiki/Historia_de_la_inteligencia_artificial

http://es.wikipedia.org/wiki/Inteligencia_artificial

http://www.ayc.unavarra.es/miguel.pagola/TEMA1IA.pdf

ALGORITMOS DE BÚSQUEDA

http://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda

http://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad

A*

http://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*

http://en.wikipedia.org/wiki/A*_search_algorithm

ETERNITY 1 Y 2

http://www.microsiervos.com/archivo/puzzles-y-rubik/eternity-ii-puzzle.html

http://es.eternityii.com/

http://su.doku.es/2007/01/26/eternity-y-eternity-ii/